EE282 : Computer Architecture and Organization

Review Session

on Programming Assignment #2

Autumn, 2001

Outline

- You’re going to build a pipelined version of a MIPS-lite machine in Verilog!
- Getting Started
- Three steps of implementing DLX pipeline

  1. Insert registers between stages
  2. Implement bypassing paths
  3. Implement interlocks

Getting Started

- Copy the code:

```
cp -r /usr/class/ee282/proj2/* .
```

- Compile the programs

```
compile282 [source file]
```

- Run the Verilog model

```
verilog -f master +waves +regs +output
```

What to Turn in Electronically

- Your Verilog code
- README describing the changes you made to the Verilog code and why. Give us your feedback on this assignment.
- GROUP file listing members of your group
- The cycle counts for the test programs:

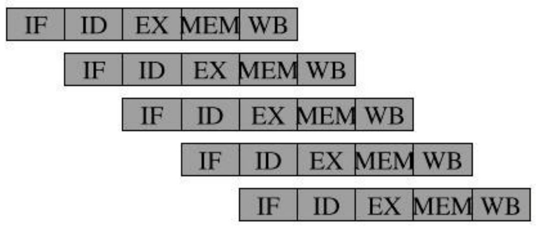
```
add.s bypass.s interlock.s
bubble.c quick.c
```

The Big Picture

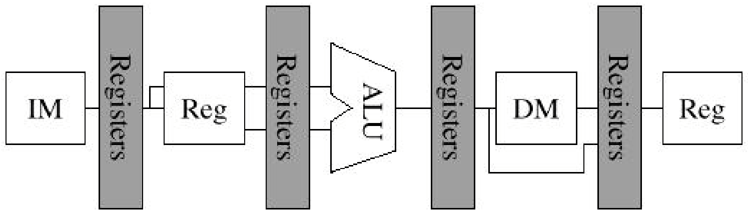
- From:
  - A multi-cycle implementation
  - Work with single-instruction at a time
  - No hazards
  - 5 stages/inst.
  - 5 stages total
- To
  - Pipelined implementation
  - Work with multiple-instructions at a time (up to 5)
  - Hazards everywhere
  - 5 stages/inst.
  - 5 stages total

The First Step: Insert Registers between Stages

- What is pipelining?
  - Exploiting instruction-level parallelism by overlapping the execution of consecutive instructions.



We need registers between stages



Refer to the textbook (Ch. 3) and the lecture notes for detailed explanation.

Verilog Model for Pipelined Registers

Is provided in ff.v

```
module propagate (in, stall, reset,
out);
```

Instantiate like this:

```
propagate #(4) demo_ff (in_state, 1'b0,
1'b0, out_state);
```

Use #(width) field to specify the bit width of the flip-flop.

Testing

- After finishing the first step, test your code with `add_nop.s.`
- `add_nop.s` doesn’t have pipeline hazards at all (it explicitly has NOPs between dependent instructions), so your code should work with it!
- If this works, great! You may wish to create your own test cases to confirm correctness.

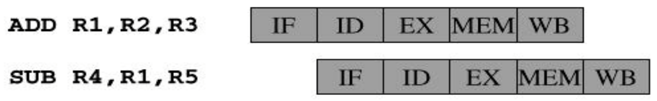
add_nop.s

```
main:
lw      $2, 0x160($0)    /* load a */
lw      $3, 0x164($0)    /* load b */
nop
nop
add     $4, $3, $2       /* c = a + b */
nop
nop
sw      $4, 0x168($0)    /* store c */
j       $31
```

The Second Step: Implement Bypassing Paths

- Our pipelined model is not yet complete. Why?
  - There are data hazards (RAW hazard).

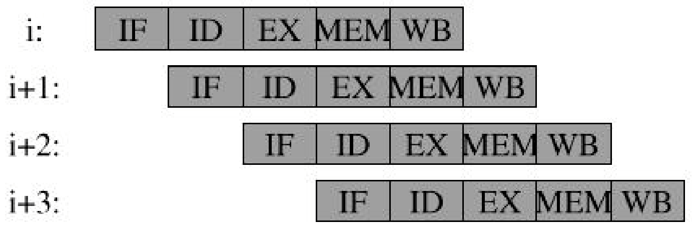
  For example,



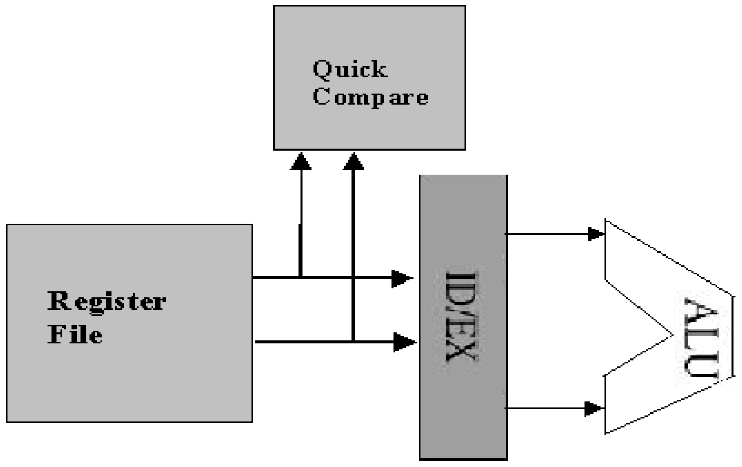
the incorrect value of R1 will be used in SUB instruction.

•How do we solve this problem? Use bypassing!!

What Bypassing Paths do we need?

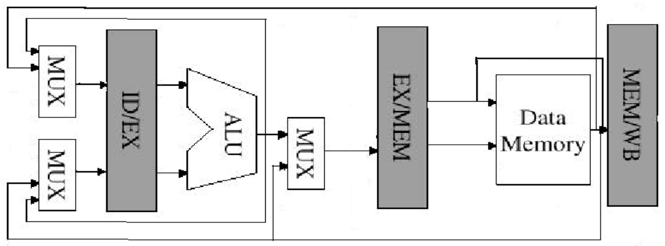


Bypassing Paths (cont'd)



How to Implement Bypass Paths

- Use multiplexors and control logic (e.g. comparators).



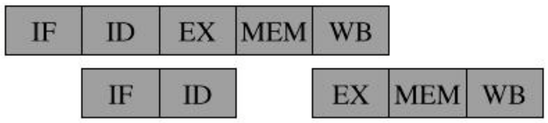
Testing Bypassing

- When the second step is done, test your code with `bypass.s`
- You may wish to create additional test cases (manually insert NOPs when needed)
- If these test programs work, go to the last step - implementing interlocks!

The Third Step: Implement Interlocks

- Our pipelined model is still not complete.
- Load interlock - we need to stall 1 clock cycle for the following sequence:

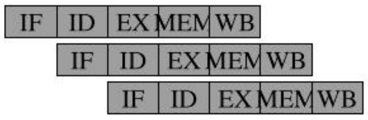
```
lw   R2, 200(R3)
add  R4, R2, R1
```



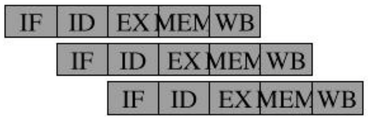
Implement Interlocks (cont'd)

- Assume predict-not-taken scheme
- Stall 1 cycle for taken branch (control hazard)

Not taken branch



Taken branch



Testing Interlocks

- If you finish all three steps, you are done!!
- Test your complete pipelined machine with the following test programs:

  `interlock.s bubble.c quick.c`
- When all these programs work, be happy.
- Follow online submit instructions.
- Get some sleep.